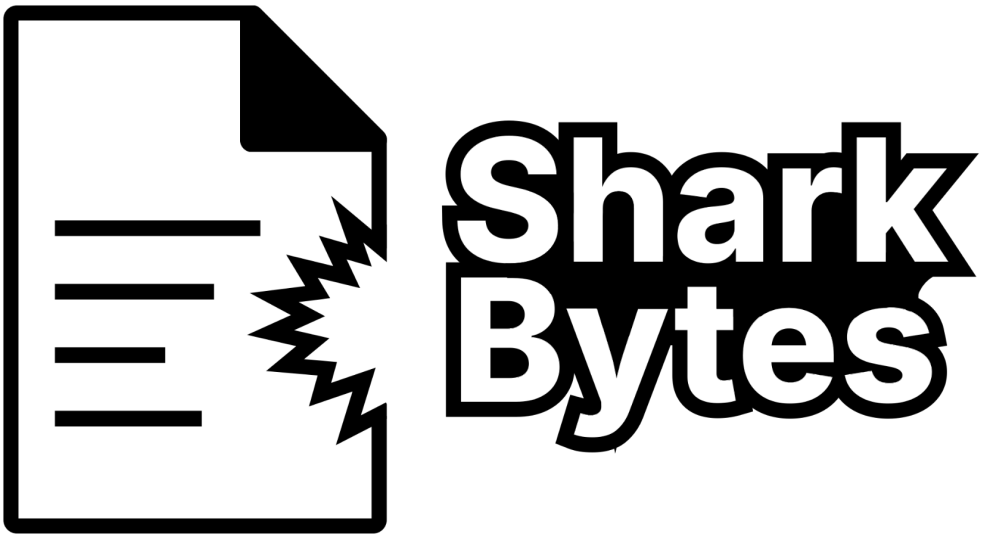
Testing Plan - 03/17/2026



Clients: David Rogowski, Mara Dzul, Pilar Rinker

Mentor: Scott LaRocca

Team: Jered Angous, Daniel Arden, Alexander King, Anthony Narvaez

The purpose of this document is to discuss how our team intends to test our application to ensure it reliably provides the users the intended functionality

Table of Contents

Introduction

The Colorado River supports over 40 million people and plays a huge role in sustaining agriculture, ecosystems, and communities across the southwest United States. To monitor these resources, different organizations such as the U.S. Geological Survey, Arizona Game and Fish Department, Grand Canyon Monitoring and Research Center, and U.S. Fish and Wildlife Service come together to collect accurate and efficient data from the river. However, the current field data entry system, called SHOALS, is outdated, difficult to use, and has a lack of documentation. To deal with these challenges, our team is creating a modern, flexible, and maintainable data entry application that is designed to improve usability, reliability, and data quality for conservation efforts.

In this document, our team will show how software testing plays a big role in ensuring that the system meets its functional and non-functional requirements. The application has several key components, such as a frontend user interface for data entry and navigation. There will be a backend that will be responsible for managing the application logic and a database for data storage. The system will also depend on external components such as the Biomark PIT scanner for automated data input and USBs as backup storage devices. The scope of testing includes all the core features that were defined in the demo flight plan, such as dynamic field generation, automatic backups, wireless scanner communication, a modernized user interface, and data exporting. These features will be tested throughout the development of the application, especially during major goals such as Alpha Demo I, Alpha Demo II, and Acceptance Demo. Out-of-scope items include any of the SHOLES legacy code, since we are starting from scratch.

To make sure our system works well, our team will use different types of testing throughout our development process. Unit testing will focus on the application's SQL, which will be tested using flutter_test regularly. Integration testing will focus on making sure core features of the application work together correctly. Usability testing will make sure that the application is easy to use for researchers in a real-world environment. Together, these testing methods will make sure that all the implemented functionality is checked before each major demonstration.

The testing strategy will focus more on the critical parts of the system, like the data handling and core features, because if errors occur in that area, it could impact data accuracy and workflow. Scanner communication and data exporting are also critical to focus on due to their higher risk of failure. Usability testing is also important since the application is used in real-world field conditions where efficiency is essential. By focusing on these areas, our team can ensure that the system that is created is reliable. The following section gives a more detailed explanation of each testing approach.

Unit Testing

Introduction

Our project is primarily designed to be an interface for performing CRUD (Create, Read, Update, Delete) operations on a database. We've determined that the critical modules in our application to unit test mostly relate to these operations. In particular, our application's SQL Insert and Update generators, and our SQL Create statement validator are our critical modules. We consider these modules to be important business logic to have high test coverage on.

Testing

We are using flutter_test as our testing framework of choice, as it is included in Flutter, is well-documented, and is simple to use. We are executing these tests on our local machines at this point in time, and ensuring they all pass before we create a pull request from our dev branches to main, not doing it automatically in CI. We are tracking test pass/fail status, and requiring 100% of our tests to pass before being merged to main.

Units Under Test

SQLInsertGenerator.generate(tableName, dataValueMap)

Purpose:

Test Case Categories:

- Valid Inputs: ValueMap has one or more values, tableName is a non-empty string
- Boundary Cases: N/A
- Invalid Inputs: ValueMap has no provided values, or tableName is an empty or invalid string

Sample Tests:

- Valid Inputs: Statement should have matching counts of columns and placeholders
    - generator.generate(table.name, table.toValueMap())
    - Regex identifies amount of columns and placeholders, test compares them to make sure they're the same
- Boundary Cases: Generated SQL statements should not have trailing commas
    - Checks to see if generated string contains a comma followed by a parenthesis, fails if so
- Invalid Inputs: Throws an exception when an empty Value Map is provided
    - generator.generate("test", <String, dynamic>{}) - Exception

SQLUpdateGenerator.generate(tableName, dataValueMap, identifier, key)

Purpose: Generates update statements to be executed on the database

Test Case Categories:

- Valid Inputs: ValueMap has one or more values, tableName is a non-empty string, has non-empty identifier and key
- Boundary Cases: N/A
- Invalid Inputs: ValueMap has no provided values, tableName is an empty or invalid string, identifier and/or key are missing

Sample Tests:

- Valid Inputs: When given all necessary parameters, the update generator creates a valid SQL string
    - With the correct parameters, the generator should return an exact string
    - Equals “UPDATE specimen SET id = ?, SET species = ? WHERE rowid = 1;” - pass
- Boundary Cases: Generated SQL statements should not have trailing commas
    - Does not contain “?, WHERE”
- Invalid Inputs: Throws an exception when an empty Value Map is provided
    - “generator.generate(table, <String, dynamic>{}, identifier, key);”

SQLSecurityValidator.isSafe(statements)

Purpose: Validates list of SQL CREATE statements to ensure they do not do anything that could damage the integrity of existing data or reduce security on the created database.

Test Case Categories:

- Valid Inputs: Valid SQL Create statements, with or without comments.
- Boundary Cases: Should accept when comments or identifiers contain disallowed keywords as long as the CREATE statement is still valid.
- Invalid Inputs: Has modifiers, PRAGMA statements, tables, select statements, delete statements, drop statements, or unterminated comments. Should still block any of them, even if they are disguised by case or interrupted by comments.

Sample Tests:

- Valid Inputs: Allows standard SQL create statement
    - “CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);” - pass
- Boundary Cases: Ignores unsafe keywords in single-line comments

    “CREATE TABLE settings (id INT);

    -- This next line is a comment and should be ignored

    -- SELECT * FROM secrets;” - pass
- Invalid Inputs: Blocks forbidden keywords in plain code
    - “CREATE TABLE x (id INT); DROP TABLE users;” - fail

These modules are the primary ones our team feels need unit testing. We do not anticipate unit testing UI logic as we are primarily utilizing UI modules and components provided by the Flutter framework itself, which are extensively tested.

SqlLexer(String sql, Set<String> keywords).tokenize()

Purpose: Turns a single SQL statement into a list of Tokens, which is then used for security validation. It allows code to identify whether a token is a keyword, identifier, symbol, et cetera, which the SQLSecurityValidator then uses to validate whether a SQL statement is safe.

Test Case Categories:

- Valid Inputs: Valid SQL
- Boundary Cases: Does not identify keywords inside string literals or comments as keywords, does not identify comment markers inside of string as comments, does not identify comment markers as comments inside of strings
- Invalid Inputs: Unterminated string literals or multi-line comments

Sample Tests:

- Valid Inputs: A valid SQL create statement is correctly tokenized
    - “CREATE TABLE users;” - keyword, keyword, identifier, symbol
- Boundary Cases: Keywords with scrambled case are still labeled as keywords by the tokenizer
    - “cReAtE tAbLe” - keyword, keyword
- Invalid Inputs: Throws on unterminated multi-line comment
    - “/* hello” - FormatException

SqlSchemaParser.getStatements(String sql)

Purpose: SqlSchemaParser strips SQL strings of comments, normalizes the case, and splits multiple SQL statements in a single string into a list of SQL statements. These statements are then tokenized in the SqlLexer and verified for safety in the SqlSecurityValidator.

Test Case Categories:

- Valid Inputs: A string containing any combination of valid Sql.
- Boundary Cases: Statements with delimiters inside of identifier strings or comments
- Invalid Inputs: Incorrectly formatted SQL, SQL missing delimiters, SQL with unterminated strings or comments.

Sample Tests:

- Valid Inputs: Correctly splits, removes comments, and normalizes two SQL statements
    - Verify 2 statements are properly split and formatted (too long to paste the test)
- Boundary Cases: Shouldn't split on quotes inside of a string
    - "CREATE TABLE "quoted;table" (id INT);" - kept as one statement
- Invalid Inputs: Handles empty inputs
    - "" - returns nothing

Integration Testing

Having considered how individual components behave, both in terms of accuracy as well as efficiency, it is important to ensure these components can work together to produce accurate results. This is exactly the type of analysis conducted in the Integration Testing phase, which focuses on ensuring the components in different layers of the application work together without producing unexpected or incorrect results.

Our approach for this phase of testing begins with identifying the integration points of the project. These are the areas of the application that directly impact the user, the data the user is working with, or the stability of the application. For our application, these integration points consist mainly of the requested features including:

1) **Dynamic Field Generation**: Allows users to define the tables and fields they expect to work with.
2) **Automatic Backups**: Allows collected data to be safely backed up on multiple external devices whenever data is recorded.
3) **Wireless Scanner Communication**: Provides an interface by which users can connect a scanner and auto-populate fields with the produced data.
4) **Modernized User Interface**: Provides an intuitive and reactive user interface
5) **Data Exporting and Synchronization**: Allows users to sync multiple external drives as well as export collected data as a CSV or TSV file.

Each of these features consists of modules and classes that span the domain, application, infrastructure, and presentation layers of the application. Therefore, in order to verify the integration functionality of these features, we must verify interoperability between these different modules and layers. For the majority of the features in this application, the flow of data follows the following pipeline:

1) The presentation layer (user interface) allows the user to perform an action (this can be connecting a scanner, selecting a drive, saving a record, etc)
2) The UI then makes use of an application layer use case. This is a class designed to serve one specific purpose and nothing else.
3) This use case ensures business data is being followed by using domain layer objects and methods. It also handles logic by calling interfaces.
4) These interfaces are classes designed to expose functionality, but abstract the implementation. They are implemented by classes in the infrastructure layer.
5) The infrastructure implementations make use of external tools and dependencies such as SQLite, Drift, Win32, and others. They provide the exact functionality promised by the interfaces.

With this pipeline in mind, we can identify that interactions between layers are crucial for ensuring that data flows consistently. We must verify that each layer can handle any errors or edge case scenarios occurring upstream and produce reliable results for any downstream modules. Similarly, database interactions are crucial because they handle the data that is used by the researchers. So all operations performed on the database must

preserve the integrity and accuracy of the data. These are the two major areas of integration testing that the application will be put through.

It is also worth noting that the environment in which these tests will be performed has to mimic both the real application's software runtime environment, as well as the actual operating environment expected to be encountered when researchers use the app. Though our clients mentioned the anticipated use of Linux systems in the future, the current machines used by our clients run Windows. This is why our testing will mostly focus on the Windows 10 and 11 operating systems and the Flutter and Dart runtime environments. Similarly, testing will simulate the creation of scan events, as well as record entries.

Testing will take place within the "/testing" directory of the application, and will make use of a temporary database for any data needs. These databases will be copies of schemas that would be compatible with the real application, and will have the data cleared between tests. Test records and scans will be generated programmatically in order to ensure both good coverage of the code as well as deterministic results which can be re-created. With this in mind, the following are some of the integration tests that are slated to take place.

Integration Point: Scanner Infrastructure - Presentation Layer

Feature: Auto Populating Fields with Scanner Data

Scenario Description: A researcher scans a PIT tag with a Bluetooth Scanner

Integration Steps:

- User scans a PIT tag with a Bluetooth Scanner
- Scanner sends the data through a virtual COM port, managed by scanner infrastructure
- Scanner infrastructure converts raw bytes to characters and creates a ScanEvent stream containing parsed data
- Riverpod Providers expose this ScanEvent stream to the User Interface
- UI verifies a scanner target field is selected
- UI pastes the scanner information into the selected field

Expected Results:

Scanner data is parsed correctly

UI successfully extracts tag information from the ScanEvent

Detected tag ID matches ID on the scanner

UI displays no processing errors

Failure Handling:

Invalid input results in a clear validation error

Not having a selected scanner target field alerts the user

Invalid tag formats are recognized and also alert the user

Integration Point: FlexField Infrastructure - Presentation Layer

Feature: Flexible Field Generation

Scenario Description: A researcher enters their project Schema

Integration Steps:

- User enters a schema representing their data collection structure via the UI
- The presentation layer calls an application layer verification use case
- The application layer calls a domain-layer validation function
- If valid, the application layer use case creates the table(s) via the sanitized schema. If not valid, the application layer rejects the request and notifies the user
- Once the schema is created, a new project option is added to the list

Expected Results:

Attributes defined in the schema appear as fields in the data entry screen

All specified tables and fields are displayed accurately

UI displays no schema processing errors

Failure Handling:

Invalid input results in a clear validation error

Invalid schemas do not create new project options

Integration Point: Auto Backups Infrastructure Layer - Application Layer

Feature: Auto backup target selection and data backup

Scenario Description: Backup targets are automatically selected when the application starts, and data is backed upon saving a record

Integration Steps:

- User opens the application, and the Presentation layer sends a selection request to the Application layer
- Application Layer calls the Infrastructure implementation for drive detection
- All currently connected drives are selected
- When the user saves new data, the presentation layer sends a backup request to the application layer use case
- The application layer calls the infrastructure backup implementation
- Infrastructure backup copies current database to external drive

Expected Results:

All drives connected on startup are automatically selected

Backed up data is accurate, and matches data on the main database

Backups occur on every record save, within 1s of saving

UI displays backup processing errors

Failure Handling:

Backup failure is clearly displayed on the User Interface

Auto selection failure should still allow for manual drive selection

Usability Testing

Usability testing for our program will look at the workflow of the users, ensuring they can efficiently use the program. This is a major component of usability testing, as we are making a replacement for a program that exists and is currently used. Ensuring a similar workflow can help reduce the time needed to learn the new application. We will also be looking to identify any confusing aspects as well as the overall user experience.

Our approach for usability testing will involve field and in-office testing to ensure features work and that the expected workflow and speed of entering data are at an acceptable level. This is a replacement for an existing program, so the users will have experience with a similar data entry program. We are going to look at how users navigate the program if the data can be entered at speeds that match the rate in the field or from the previous program.

- **Field Testing:** The application will be used directly in the field in the Colorado River, where users will perform actions like scanning and entering data. This will help ensure the system performs reliably in real-world use and supports efficient data entry.
- **In Office:** The application will be brought to the office of the agencies we are working with to use it in a controlled environment. This allows us to closely observe user interactions, identify usability issues, and gather direct feedback. We will measure factors such as data entry time, accuracy, and ease of use.

We will have our usability testing during phase three, which starts right after the alpha demo II. We have an in-office test before we do our field test. We also plan to have at least one more office test after the field to ensure any changes we have to make after the field, as well as ensure the program is acceptable to our clients before delivering it to them.

Testing Workflow & Quality Control

Our testing workflow is primarily local, as we run our tests and ensure they all pass before making a pull request to main. If any tests fail, the developer on that branch should attempt to fix them before making a pull. If additional testing is performed later and bugs are identified, they should take priority over the development of additional functionality. The application should have a strong foundation, and technical debt should be eliminated early to prevent it from building over time.

As we attempt to fix bugs as we discover them, we opt not to use severity levels in most cases. However, if multiple bugs are discovered, our team will discuss internally which bug should be prioritized and assign it to someone to fix it.

We don't consider bug fixes to be part of a feature development cycle. Bugs should take priority over feature development. Bugs should be fixed as fast as they can. We prefer not to include any bugs in our alpha demonstrations. However, as alpha demonstrations are put on tight deadlines, sometimes bugs make their way in. They should be eliminated as soon as possible after the alpha demonstration concludes.

The only acceptable issues we tolerate are user interface bugs that don't drastically affect the user experience. As the capstone project takes place over a short period of time, compromises such as accepting small bugs in order to meet more of the required functionality of the team.

Conclusion

In conclusion, this document outlines the approach we will be using to ensure our application is reliable through unit, integration, and usability testing. Testing will be done throughout the development process, with a major focus during Alpha Demo I, Alpha Demo II, and the Acceptance Demo. Extra attention will be given to important areas of the application, such as data accuracy and usability, so that the app works well. By focusing on the key areas and testing the application regularly, our team will achieve our goal of creating a system that is reliable, works properly, and is easy to use for users in the field.